Développement d'un logiciel de messagerie instantanée avec Dotnet (version simplifiée)

Propriétés	Description
Intitulé long	Développement d'un logiciel de messagerie instantanée
Formation concernée	BTS Services informatiques aux organisations
Matière	SLAM4 - Réalisation et maintenance de composants logiciels
Présentation	Développement d'un logiciel de messagerie instantanée. La communication est basée sur les sockets entre deux applications dotnet.
Notions	▪ Programmation à l'aide d'objets ▪ Développement orienté réseau
Transversalité	Les protocoles réseau
Pré-requis	Le langage C#, les applications WinForms.
Outils	Les applications fournies ont été réalisées avec Visual Studio 2008 et sont destinées au framework dotnet 3.5, elles peuvent facilement être recompilées avec un autre outil et/ou cibler une autre version du framework. - CoursSockets.doc : support de cours élève. - Exemple01 : un premier exemple d'application simple. - Exemple02 : exemple de réception asynchrone. - Exemple03 : utilisation d'un objet *BackgroundWorker*. - Chat : application simple de chat.
Mots-clés	C # , DotNet
Durée	10 heures
Auteur(es)	Pierre Loisel
Version	v 1.0
Date de publication	Novembre 2008

Ce document n'est pas une étude exhaustive du domaine. Il présente simplement une manière de réaliser des applications communicantes en C#.

L'exécution d'une application utilisant les sockets à partir d'un disque réseau peut poser des problèmes de sécurité. Si vous obtenez une exception concernant la sécurité, configurez les paramètres de sécurité du framework. Vous pouvez consulter la page *http://msdn.microsoft.com/fr-fr/library/2bc0cxhc.aspx* pour obtenir des informations concernant l'outil .NET Framework Configuration (Mscorcfg.msc).

A/ Présentation

1. Les sockets

Les sockets fournissent un mécanisme générique de communication entre les processus. Elles sont apparues pour la première fois en 1986 dans la version UNIX de l'université de Berkeley.

Un processus peut être sommairement défini comme une application (ou un service) en cours d'exécution.

Un socket permet à un processus (le client) d'envoyer un message à un autre processus (le serveur). Le serveur qui reçoit ce message peut alors accomplir toutes sortes de tâche et éventuellement retourner un résultat au processus client.

Lors de la création d'un socket, il faut préciser le type d'adressage, le type de message et le protocole transport utilisés. Nous utiliseront : IPV4, les datagrammes simples et le protocole UDP.

2. Point de terminaison

Un point de terminaison (*EndPoint*) est défini par une adresse IP et un numéro de port. Une communication s'établit entre deux points de terminaison.

Un processus serveur reçoit les messages destinés à un numéro de port et à un protocole transport (UDP, TCP, ...) déterminés. Un client désirant envoyer un message à un serveur doit donc créer un point de terminaison représentant le récepteur du message en fournissant l'adresse IP du serveur et le numéro de port écouté.

Pour envoyer un message, un processus doit également créer un point de terminaison représentant l'émetteur du message en fournissant sa propre adresse IP. Il ne fournit pas de numéro de port, c'est le système qui attribuera un numéro de port libre pour la communication.

3. Principe de communication

Lors de l'envoi d'un message, il faut :

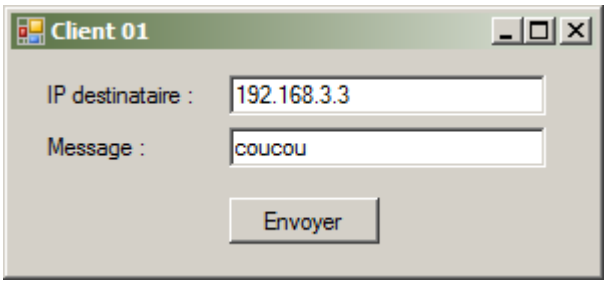
- Créer un socket,
- Créer le point de terminaison émetteur,
- Lier le socket au point de terminaison émetteur (ce qui précisera le protocole transport utilisé pour l'émission).
- Créer le point de terminaison récepteur,
- Envoyer le message à l'aide du socket vers le point de terminaison récepteur.

Pour recevoir un message, le processus serveur doit :

- Créer un socket,
- Créer le point de terminaison récepteur (lui-même donc),
- Lier le socket au point de terminaison récepteur (ce qui précisera le protocole transport utilisé pour la réception).
- Créer le point de terminaison émetteur (sans préciser l'adresse IP ni le numéro de port puisqu'ils ne sont pas connus à ce stade).
- Mettre le socket en état de réception en lui fournissant un buffer pour les données à recevoir, et une référence au point de terminaison émetteur.
- Lorsque le message est effectivement reçu, le point de terminaison émetteur est renseigné.

4. Premier exemple (exemple01)

Le client :




```
public Fm_client()
{
    CheckForIllegalCrossThreadCalls = false;
    InitializeComponent();
    adrIpLocale = getAdrIpLocaleV4();
}
private IPAddress adrIpLocale;
private IPAddress getAdrIpLocaleV4()
{
    string hote = Dns.GetHostName();
    IPHostEntry ipLocales = Dns.GetHostEntry(hote);
    foreach (IPAddress ip in ipLocales.AddressList)
    {
      if (ip.AddressFamily == AddressFamily.InterNetwork)
      {
        return ip;
      }
    }
    return null; // aucune adresse IP V4
}
private void bt_envoyer_Click(object sender, EventArgs e)
{
    byte[] message;
    Socket sock = new Socket( AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);
    IPEndPoint epEmetteur = new IPEndPoint(adrIpLocale, 0);
    sock.Bind(epEmetteur);
    IPEndPoint epRecepteur = new IPEndPoint(
                    IPAddress.Parse(tb_ipDestinataire.Text), 33000);
    message = Encoding.Unicode.GetBytes(tb_message.Text);
    sock.SendTo(message, epRecepteur);
    sock.Close();
}
```

Remarque : la chaîne à envoyer est transformée en un tableau de bytes.

Le serveur :


```
public Fm_serveur()
{
   CheckForIllegalCrossThreadCalls = false;
   InitializeComponent();
   adrIpLocale = getAdrIpLocaleV4();
}
private IPAddress adrIpLocale;
private IPAddress getAdrIpLocaleV4()
{
   string hote = Dns.GetHostName();
   IPHostEntry ipLocales = Dns.GetHostEntry(hote);
   foreach (IPAddress ip in ipLocales.AddressList)
   {
     if (ip.AddressFamily == AddressFamily.InterNetwork)
     {
       return ip;
     }
   }
   return null; // aucune adresse IP V4
}
private void bt_recevoir_Click(object sender, EventArgs e)
{
   byte[] message = new byte[40];
   Socket sock = new Socket( AddressFamily.InterNetwork,
                             SocketType.Dgram, ProtocolType.Udp);
   IPEndPoint epRecepteur = new IPEndPoint(adrIpLocale, 33000);
   sock.Bind(epRecepteur);
   EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
   sock.ReceiveFrom(message, ref epTemp);
   IPEndPoint epEmetteur = (IPEndPoint)epTemp;
   string strMessage = Encoding.Unicode.GetString(message);
   MessageBox.Show( epEmetteur.Address.ToString() + " -> "
                   + strMessage);
}
```

Remarques :

- La fonction *getAdrIpLocaleV4* retourne l'adresse IP V4 de l'hôte en utilisant le service DNS. Le nom de l'hôte local est récupéré en utilisant le DNS.
- Elle est appelée dans le constructeur du formulaire.
- La méthode *ReceiveFrom* de la classe Socket attend un paramètre de type EndPoint. Il faut donc opérer une conversion de type pour récupérer le point de terminaison émetteur.
- Le message est un tableau de bytes. Il faut le transformer en chaîne de caractères.
- L'instruction « CheckForIllegalCrossThreadCalls = false; » dans le constructeur des deux formulaires permet de s'affranchir simplement des problèmes liés à la mise à jour de l'interface lors de la réception d'un message.

Mise en œuvre :

- Créer l'application client et l'application serveur.
- Lancer les deux applications (sur le même poste ou sur deux postes différents).
- Cliquer sur le bouton *recevoir* du serveur.
- Renseigner l'adresse IP du serveur et le message à envoyer dans l'application client.
- Cliquer sur le bouton *envoyer* du client.
- Le serveur affiche l'adresse IP et le message du client.

B/ Réception asynchrone

1. Principe

Lorsque l'on clique sur le bouton *recevoir* du serveur, celui-ci se trouve bloqué en attente du message et ne peut accomplir aucune autre tâche. Pour y remédier, il est possible d'utiliser le mécanisme de réception asynchrone fourni par Dotnet. Ce mécanisme est basé sur les threads. Un thread peut être vu comme un « mini processus » interne à une application. Une application peut avoir plusieurs threads et donc exécuter plusieurs tâches simultanément (cette simultanéité n'est que fictive, en tous cas sur une machine mono-processeur).

Le principe est le suivant :

> Mettre le socket en état de réception en utilisant la méthode ci-dessous :

```
sock.BeginReceiveFrom(message,0,40,   SocketFlags.None, ref epTemp,
                                      new AsyncCallback(recevoir),null);
```

- Le socket est mis en état de réception dans un thread de réception différent du thread principal. Le thread principal poursuit immédiatement son exécution, le programme n'est pas bloqué par cet appel.
- L'avant dernier paramètre indique que la méthode *recevoir* doit être appelée dès la réception d'un message par le thread de réception.

> Gérer le message reçu :

Quand le thread de réception reçoit le message, il le prend en charge (l'affiche par exemple), et se termine. L'idéal est de remettre le socket en état de réception immédiatement, de manière à accepter plusieurs messages les uns après les autres.

2. Modification du serveur (exemple02)


```
public Fm_serveur()
{
    CheckForIllegalCrossThreadCalls = false;
    InitializeComponent();
    init();
}
private int lgMessage = 40;
private IPAddress adrIpLocale;
private Socket sock;
private IPEndPoint epRecepteur;
byte[] message;
private IPAddress getAdrIpLocaleV4()
{
    // Idem version pécédente
}
private void init()
{
    message = new byte[lgMessage];
    adrIpLocale = getAdrIpLocaleV4();
    sock = new Socket(AddressFamily.InterNetwork, SocketType.Dgram,
ProtocolType.Udp);
    epRecepteur = new IPEndPoint(adrIpLocale, 33000);
    sock.Bind(epRecepteur);
    EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
    sock.BeginReceiveFrom(  message, 0, lgMessage, SocketFlags.None,
                            ref epTemp, new AsyncCallback(recevoir),
                            null);
}
private void recevoir(IAsyncResult AR)
{
    EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
    sock.EndReceiveFrom(AR, ref epTemp);
    IPEndPoint epEmetteur = (IPEndPoint)epTemp;
    string strMessage;
    strMessage = Encoding.Unicode.GetString(message, 0, message.Length);
    Array.Clear(message, 0, message.Length);
    sock.BeginReceiveFrom(  message, 0, lgMessage, SocketFlags.None,
                            ref epTemp, new AsyncCallback(recevoir),
                            null);
    MessageBox.Show( epEmetteur.Address.ToString()
                    + " -> " + strMessage);
}
```

Remarques :

- Le serveur se met en réception dès le lancement de l'application, il n'y a donc plus besoin d'un bouton *recevoir*.
- Dès la réception d'un message, le serveur relance un nouveau thread de réception.
- La réception ne s'arrête qu'à la fermeture du programme.
- Vous pouvez tester l'envoi de messages à partir de deux clients différents.

C/ Un exemple plus complet

1. Améliorer l'interface

Modifiez votre application serveur de la manière suivante :

- Ajouter un contrôle de type *ListBox* (lb_messages) sur le formulaire.
- Au lieu d'afficher le message reçu, la méthode *recevoir* doit maintenant ajouter ce message à la liste *lb_messages*.
- Ajouter une propriété *nbMessages* à la classe *Fm_serveur*. Cette propriété sera destinée à compter le nombre de messages reçus.
- Ce nombre de messages sera affiché dans un label *lbl_nbMessages*.

2. Gérer un message plus complexe

Le client va maintenant envoyer un message constitué de son adresse IP et du texte du message lui-même. Une solution simple consiste à envoyer une chaîne contenant les deux éléments séparés par un caractère particulier.

Exemple : « 192.168.200.3#Le texte du message# ».

A la réception de cette chaîne, le serveur doit retrouver les deux éléments du message. Il peut le faire de la manière suivante :

- Récupérer le message reçu sous la forme d'une chaîne de caractères :

```
strMessage = Encoding.Unicode.GetString(message, 0, message.Length);
```

- Isoler les différents elements dans un tableau de chaînes :

```
string[] tabElements;
char[] separateur = new char[1];
separateur[0] = '#';
tabElements = strMessage.Split(separateur);
```

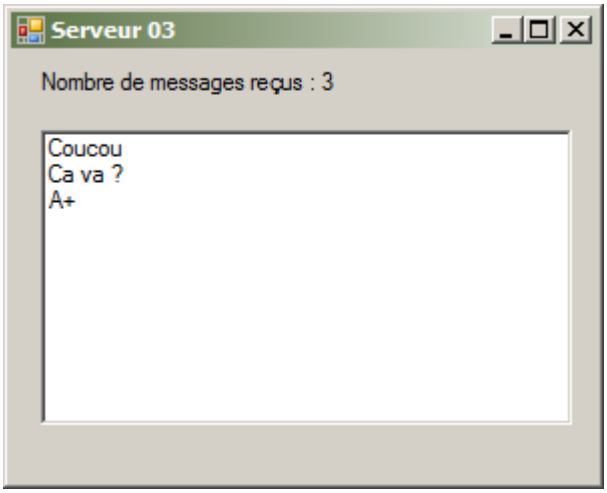
- tabElements[0] contient l'adresse IP.

- tabElements[1] contient le texte du message.

Remarques :

- La méthode Split découpe la chaîne strMessage en plusieurs éléments et place ces éléments dans le tableau tabElements.
- Elle prend en paramètre un tableau contenant les caractères jouant le rôle de séparateur pour découper la chaîne strMessage.
- Il faut donc que ce tableau contienne le caractère '#'.

3. Modification du serveur (exemple03)




```
private int nbMessages;
// idem …
private void init()
{
    nbMessages = 0;
    // idem …
}
private void recevoir(IAsyncResult AR)
{
     EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
     sock.EndReceiveFrom(AR, ref epTemp);
     string strMessage;
     strMessage = Encoding.Unicode.GetString(     message, 0,

    message.Length);
     string[] tabElements;
     char[] separateur = new char[1];
     separateur[0] = '#';
     tabElements = strMessage.Split(separateur);
     nbMessages++;
     string affichage = tabElements[0] + " -> " + tabElements[1];
     lb_messages.Items.Add(affichage);
     lbl_nbMessages.Text =  "Nombre de messages reçus : "
                            + nbMessages.ToString();
     Array.Clear(message, 0, message.Length);
     sock.BeginReceiveFrom( message, 0, lgMessage, SocketFlags.None,
                            ref epTemp, new AsyncCallback(recevoir),
                            null);
}
```

D/ Une vraie conversation

1. Transmettre un message à tout le monde

Pour transmettre un message à plusieurs hôtes, il y a au moins deux solutions :

- Transmettre le même message successivement aux différents destinataires, ce qui nécessite une simple boucle.
- Envoyer le message en broadcast.

Pour envoyer un message en broadcast, il suffit de modifier ainsi la procédure d'envoi :

```
private void bt_envoyer_Click(object sender, EventArgs e)
{
    byte[] messageBytes;
    Socket sock = new Socket( AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);
    sock.SetSocketOption(SocketOptionLevel.Socket,
                         SocketOptionName.Broadcast, true);
    IPEndPoint epEmetteur = new IPEndPoint(adrIpLocale, 0);
    sock.Bind(epEmetteur);
    IPEndPoint epRecepteur = new IPEndPoint(IPAddress.Broadcast, 33000);
    string leMessage = "Contenu du message";
    messageBytes = leMessage.GetInfos();
    sock.SendTo(messageBytes, epRecepteur);
    sock.Close();
}
```

Le message sera reçu par tous les hôtes en état de réception sur le port UDP 33000.

2. Définir un protocole

Il s'agit de fixer les règles de la communication entre les différentes applications. Imaginons l'exemple d'un pauvre chat :

- Nous avons un serveur et plusieurs clients.
- Lorsqu'un client se connecte, il en informe le serveur.
- Le serveur conserve la liste des clients connectés.
- Lorsqu'un client envoie un message au serveur, celui-ci le transmet à l'ensemble des clients.
- Lorsqu'un client se déconnecte, il en informe le serveur.
- Les clients sont repérés par leur adresse IP, l'unicité des pseudos n'est pas assurée.

Nous avons besoin de définir plusieurs types de messages :

- Connexion (type C)
  Emetteur : un client qui se connecte
  Récepteur : le serveur
  Contenu : L'adresse IP et le pseudo du client
  Réaction du serveur : mémorisation du pseudo

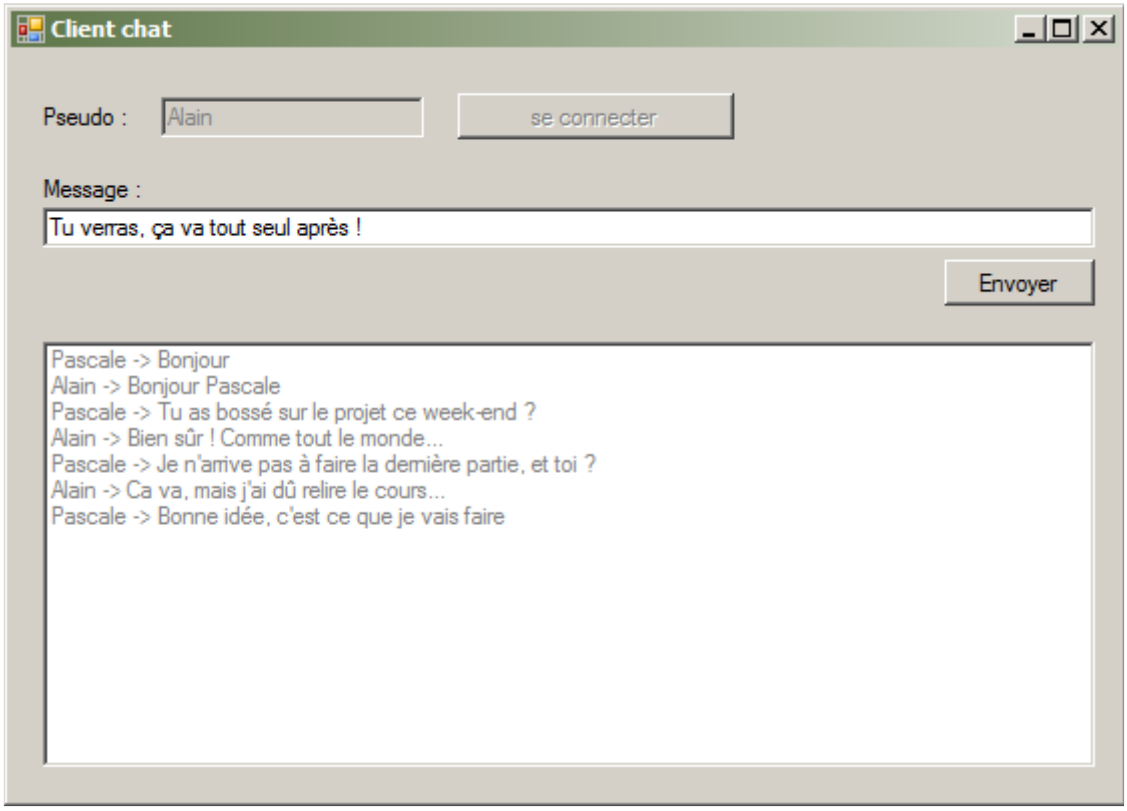
- Envoi (type E)
  Emetteur : un client qui envoi un texte sur le chat
  Récepteur : le serveur
  Contenu : l'adresse IP du client et le texte envoyé
  Réaction du serveur : réémission du texte à tous les clients

- Réémission (type R)
  Emetteur : le serveur
  Récepteur : les clients
  Contenu : l'adresse IP du serveur, le pseudo de l'auteur du texte et le texte lui-même
  Réaction des clients : affichage du pseudo et du texte

- Déconnexion (type D)
  Emetteur : un client qui se déconnecte, c'est-à-dire qui quitte l'application
  Récepteur : le serveur
  Contenu : l'adresse IP du client et son pseudo
  Réaction du serveur : mise à jour de la liste des clients connectés

E/ Le pauvre chat (Chat)

1. Le client



Généralités :
- L'adresse IP du serveur est supposée connue.
- Il faut également déterminer les ports utilisés par le serveur et par les clients.

```
public partial class Fm_client : Form
{
    public Fm_client()
    {
       CheckForIllegalCrossThreadCalls = false;
       InitializeComponent();
       adrIpLocale = getAdrIpLocaleV4();
       separateur = new char[1];
       separateur[0] = '#';
    }
    private IPAddress adrIpLocale;
    private char[] separateur;
    private IPAddress ipServeur = IPAddress.Parse("192.168.3.3");
    private int portServeur = 33000;
    private int portClient = 33001;
    private int lgMessage = 1000;
    private Socket sockReception;
    private IPEndPoint epRecepteur;
    byte[] messageBytes;
    private IPAddress getAdrIpLocaleV4()
    {
       // idem…
    }
```

L'envoi d'un message ne pose pas de problème particulier.
Chaque message est constitué des éléments suivants :

- Le type de message (C, E, R, D). Le client peut envoyer des messages C, E ou D.
- Le pseudo à l'origine du message.
- Le texte du message.

```
private void envoyer(string typeMessage,string texte)
{
    byte[] messageBytes;
    Socket sock = new Socket( AddressFamily.InterNetwork,
                              SocketType.Dgram, ProtocolType.Udp);
    IPEndPoint epEmetteur = new IPEndPoint(adrIpLocale, 0);
    sock.Bind(epEmetteur);
    IPEndPoint epRecepteur = new IPEndPoint(ipServeur, portServeur);
    string infos =   typeMessage + "#" + tb_pseudo.Text
                   + "#" + texte + "#";
    messageBytes = Encoding.Unicode.GetBytes(infos);
    sock.SendTo(messageBytes, epRecepteur);
    sock.Close();
    tb_message.Clear();
    tb_message.Focus();
}
```

Quand l'utilisateur clique sur « se connecter », le message de connexion est envoyé au serveur (il ne contient pas de texte) et le client se met en état de réception. Il devient ensuite possible de poster un texte sur le chat.

```
private void bt_connecter_Click(object sender, EventArgs e)
{
   if (tb_pseudo.Text != "")
   {
     bt_connecter.Enabled = false;
     tb_pseudo.Enabled = false;
     envoyer("C", "");
     bt_envoyer.Enabled = true;
     tb_message.Enabled = true;
     initReception();
     tb_message.Focus();
   }
}
private void initReception()
{
   messageBytes = new byte[lgMessage];
   sockReception = new Socket( AddressFamily.InterNetwork,
                               SocketType.Dgram, ProtocolType.Udp);
   epRecepteur = new IPEndPoint(adrIpLocale, portClient);
   sockReception.Bind(epRecepteur);
   EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
   sockReception.BeginReceiveFrom(   messageBytes, 0,
                                     lgMessage, SocketFlags.None,
                                     ref epTemp,
                                     new AsyncCallback(recevoir),
null);
   }
```

A la réception d'un message, le client met à jour son interface.

```
private void recevoir(IAsyncResult AR)
{
   EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
   sockReception.EndReceiveFrom(AR, ref epTemp);
   string strMessage = Encoding.Unicode.GetString(     messageBytes, 0,

   messageBytes.Length);
   string[] tabElements;
   tabElements = strMessage.Split(separateur);
   switch (tabElements[0])
   {
     case "R":
       lb_messages.Items.Add(  tabElements[1]
                               + " -> " + tabElements[2]);
       break;
   }
   Array.Clear(messageBytes, 0, messageBytes.Length);
   sockReception.BeginReceiveFrom(   messageBytes, 0,
                                     lgMessage, SocketFlags.None,
                                     ref epTemp,
                                     new AsyncCallback(recevoir),
null);
}
```

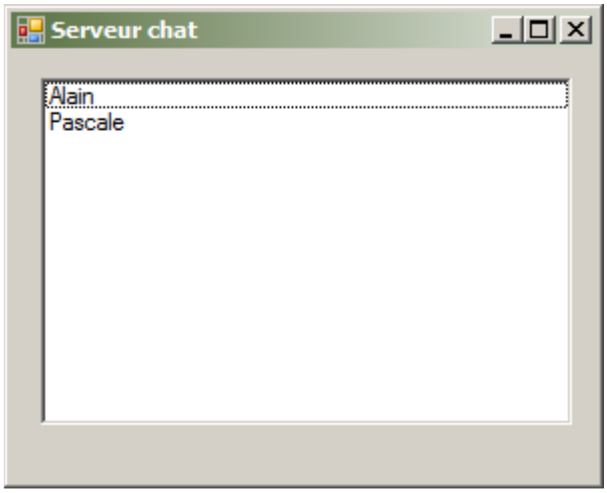
Quand l'utilisateur clique sur le bouton « envoyer », le texte saisi est envoyé au serveur.

```
private void bt_envoyer_Click(object sender, EventArgs e)
{
   envoyer("E", tb_message.Text);
}
```

La déconnexion est gérée par un message envoyé à la fermeture du formulaire.

```
private void Fm_client_FormClosing(   object sender,
                                      FormClosingEventArgs e)
{
   envoyer("D", "");
}
```

2. Le serveur



Dès son lancement, le serveur se met en état de réception sur son port.

```
public partial class Fm_serveur : Form
{
   public Fm_serveur()
   {
     CheckForIllegalCrossThreadCalls = false;
     InitializeComponent();
     initReception();
   }
private int lgMessage = 1000;
private int portServeur = 33000;
private int portClient = 33001;
private IPAddress adrIpLocale;
private char[] separateur;
private Socket sockReception;
private IPEndPoint epRecepteur;
byte[] messageBytes;
private IPAddress getAdrIpLocaleV4()
{
   // idem…
}
```

```
private void initReception()
{
   messageBytes = new byte[lgMessage];
   adrIpLocale = getAdrIpLocaleV4();
   separateur = new char[1];
   separateur[0] = '#';
   sockReception = new Socket( AddressFamily.InterNetwork,
                               SocketType.Dgram, ProtocolType.Udp);
   epRecepteur = new IPEndPoint(adrIpLocale, portServeur);
   sockReception.Bind(epRecepteur);
   EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
   sockReception.BeginReceiveFrom(   messageBytes, 0, lgMessage,
                                     SocketFlags.None,
                                     ref epTemp,
                                     new AsyncCallback(recevoir),
null);
}
```

Une méthode *envoyerBroadcast* permet l'envoi d'un message sur l'ensemble du réseau.

```
private void envoyerBroadcast(string pseudo,string texte)
{
   byte[] messageBroadcast;
   Socket sockEmission = new Socket( AddressFamily.InterNetwork,
                                     SocketType.Dgram,
                                     ProtocolType.Udp);
   sockEmission.SetSocketOption(  SocketOptionLevel.Socket,
                                  SocketOptionName.Broadcast, true);
   IPEndPoint epEmetteur = new IPEndPoint(adrIpLocale, 0);
   sockEmission.Bind(epEmetteur);
   IPEndPoint epRecepteur = new IPEndPoint(IPAddress.Broadcast,
                                                      portClient);
   string strMessage = "R" + "#" + pseudo + "#" + texte + "#";
   messageBroadcast = Encoding.Unicode.GetBytes(strMessage);
   sockEmission.SendTo(messageBroadcast, epRecepteur);
   sockEmission.Close();
}
```

Les messages reçus sont traités en fonction de leur type.

```
    private void recevoir(IAsyncResult AR)
    {
       EndPoint epTemp = (EndPoint)new IPEndPoint(IPAddress.Any, 0);
       sockReception.EndReceiveFrom(AR, ref epTemp);
       string strMessage = Encoding.Unicode.GetString(     messageBytes, 0,

       messageBytes.Length);
       string[] tabElements;
       tabElements = strMessage.Split(separateur);
       switch (tabElements[0])
       {
         case "C":
           lb_clients.Items.Add(tabElements[1]);
           break;
         case "E":
           envoyerBroadcast(tabElements[1], tabElements[2]);
           break;
         case "D":
           lb_clients.Items.Remove(tabElements[1]);
           break;
       }
       Array.Clear(messageBytes, 0, messageBytes.Length);
       sockReception.BeginReceiveFrom(    messageBytes, 0, lgMessage,
                                          SocketFlags.None,
                                          ref epTemp,
                                          new AsyncCallback(recevoir),
    null);
    }
```

Finalement, il a du chien notre pauvre chat !